AD-A080 360 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
MIME: MICROPROGRAMMABLE MINICOMPUTER EMULATOR. PHASE II. VOLUME--ETC(U)
DEC 79 T R HOYT, D A MYERS
UNCLASSIFIED AFIT/GCS/EE/79-11-VOL-I NL

1 OF 1
AD A
080360

END
DATE
FILMED
3 – 80
DDC

AFIT/GCS/EE/79-11

(9) Master's thesis

(6) MIME: MICROPROGRAMMABLE MINICOMPUTER

EMULATOR. PHASE II. [illegible]

VOLUME I. A055 [illegible]

THESIS

(14) AFIT/GCS/EE/79-11 VOL-I

(10) Thomas R./Hoyt

[redacted] [redacted]

Dean A./Myers

Captain USAF

(11) Dec 79

(12) 94

012 225

AFIT/GCS/EE/79-11

MIME: MICROPROGRAMMABLE MINICOMPUTER EMULATOR PHASE II VOLUME I

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University (ATC)
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by
Thomas R. Hoyt, B.S.M.E.
Captain USAF
Graduate Electrical Engineering
December 1979

Dean A. Myers, B.S.E.E.
Captain USAF
Graduate Computer Systems
December 1979

Acknowledgements

We gratefully acknowledge the assistance of Capt Rick Purvis, one of the creators of the original MIME, and our point of contact with ASD/ENAIB, the co-sponsor of this thesis. Rick was our mentor in the early days of the investigation, our quality control throughout the effort, and often the person with a solution when we were stymied by a problem. Without his help, this thesis would never have progressed as well nor as far.

We are indebted to the AFIT laboratory staff, led by Mr. Bob Durham. Special recognition goes to Mr. Dan Zambon, who accomplished the electrical fabrication of the enhancements that we made to MIME.

Our thesis committee, consisting of Maj Alan A. Ross, Dr. Gary B. Lamont, and Dr. Thomas C. Hartrum, performed an invaluable function with their judicious level of direction to this project. We found it most instructional and satisfying to be encouraged to make our own design decisions, as would be the case in a production environment.

Last, but assuredly not least, we thank Ms. Laura Tucker who, besides being the loving wife and source of inspiration to one of us, made extraordinary efforts to complete the typing of this lengthy report.

Contents

Volume I

List of Tables

List of Figures

List of Terms

AB	Address Bus
ALAT	A address latch
A latch	A address latch
ALB	A less than B (Comparison of bytes A and B)
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
Am2900	AMD 2900 series of integrated circuits
AMUX	A address mux
AUX	Auxiliary Module
AUXn	Auxiliary Register number n (AUX)
backplane	Wiring between edge connectors
BAR	Base Address Register (I/O)
BLA	B less than A (Comparison of bytes A and B)
BLAT	B address latch
B latch	B address latch
BMUX	B address mux
BP	Breakpoint
BRMUX	Branch mux
bus	Collection of paths for signals that perform similar functions
CCR	Condition Code Register (macro level)
CCU	Computer Control Unit
CS	Control Store
CSB	Control Store Buffer
DB	Data Bus
DBR	Data Buffer Register
DC	Display Code

DFF	Direction Flip-flop (I/O)
DMA	Direct Memory Access
DO	Derived Operand
EPROM	Erasable programmable read-only memory
FP	Front Panel
FR	Fault Register (AUX)
hex	Hexadecimal
i	Subscript used to indicate integer
IC	Instruction Counter
IM	Interrupt Mask
IMUX	Instruction mux; chooses input to 2901's
I/O	Input/Output Module
IOB	Input/Output Buffer (I/O)
IOBR	Input/Output Buffer Register (I/O)
IR	Instruction Register (CCU)
K	1024_{10}, as in 1K words = 1024_{10} words
KYBRD	Keyboard
L.S.	Least Significant
MAB	Microaddress Bus
machine level	Designation of machine language level of operation
macro	Same as machine level
MAR	Memory Address Register (MEM_1)
MB	Memory Bus
MBR	Memory Buffer Register (ALU)
MCCR	Micro Condition Code Register (ALU)
MDB	Microdata Bus
MEM1	Memory module; main program storage

MEMFF	Memory flip-flop; specifies read or write
micro	The microprogram's level of operation
microword	Group of bits used to generate control signals
MIME	Microprogrammable Minicomputer Emulator
MR	Mask Register (in Am2914)
MLC	Micro Loop Counter (CCU)
MPMUX	Mapping mux
MPROM	Mapping PROM
MSKB	Mask Buffer; buffers PL 63-56 to DB
M.S.	Most Significant
MSP	Miscellaneous Signal Panel
MTCM	Micro Test Condition Mux (ALU)
mux	Multiplexer
n	Variable used to indicate an integer number
N.A.	Not Applicable
opcode	Operation Code
PC	Program Counter
PI	Pending Interrupt
PL	Pipeline Register (CCU)
POLMUX	Polarity Mux (CCU)
PROM	Programmable read-only memory
Q	Q register in Am2901's
RA, RB	One of 16 general purpose registers in MIL-STD-1750
RA+n	Register numbered as RA + n
RAM	Random Access Memory
Ri	One of 16 2-port RAM locations in Am2901's

RQ	MIL-STD-1750 intermediate registers for multiplication and division
RQ+n	Register numbered as RQ + n
RX	Index register
SR	Status Register (Am2914)
SW	Status Word (AUX)
TCM	Test Condition Mux (ALU)
TCRB	Test Condition Register Buffer (ALU)
Timer A	MIL-STD-1750 programmable timer (AUX)
Timer B	MIL-STD-1750 programmable timer (AUX)
UART	Universal Asynchronous Receiver Transmitter
UDACi	User Definable Auxiliary Command number i
UDAFi	User Definable Auxiliary Function number i
UDTCi	User Definable Test Condition number i
WCR	Word Count Register (I/O)
YBFR	y Buffer; buffers output of Am2901's to DB

AFIT/GE/EE/79D-11

Abstract

This report describes software and hardware enhancements made to an existing educationally oriented microprogrammable minicomputer (MIME). Hardware changes included extensions of the machine architecture by incorporation of user-definable fields in the microword, corresponding to user-definable capabilities in the hardware. A microprogramming language was developed for the machine, as well as a translator. The instruction set of MIL-STD-1750, Airborne Computer Instruction Set Architecture was microprogrammed to demonstrate the utility of the translator and the processor. MIL-STD-1750 peculiar hardware was interfaced through the user-definable portion of the architecture.

The results indicated that a minicomputer instruction set could be successfully emulated using MIME. The final system of hardware, software, and documentation provides a valuable educational tool for studies of microprogramming, emulation, and computer control.

MIME: MICROPROGRAMMABLE MINICOMPUTER EMULATOR, PHASE II

I. Introduction

As indicated by the title, this investigation is a continuation of a previous effort (Ref 1). One result of that effort was the Microprogrammable Minicomputer Emulator (MIME), which was designed as a pedagogical and research tool for use in laboratory studies of microprogramming, emulation, and computer control. At the end of that investigation, referred to in this report as Phase I, the MIME hardware was built but not debugged, and MIME had never been used to execute actual machine instructions for any minicomputer. During this investigation, the hardware was made to operate correctly, user tools and documentation were generated, and a minicomputer instruction set was implemented. Since the words "microprogramming" and "emulation" often give rise to confusion, they are defined below as used in the context of this report.

Microprogramming

In any processor, proper translation of a machine instruction into a machine operation is dependent upon the application of control signals in the correct sequence to effect information transfers between processor elements. The most common methods for providing these control signals are random logic and microprogramming.

Using random logic the machine instructions are decoded by hard-wired combinatorial and synchronous sequential logic to provide the needed control signals. Thus, the machine's response to a given machine instruction is fixed and only modifiable by making hardware changes. The lowest level of control allowed the user is machine language instructions, each of which may involve many data transfers. This control at the machine language level is referred to in this report as macroprogramming or machine level programming.

In a microprogrammable machine, by contrast, the control signals are activated by groups of bits called microwords, which are stored in a special memory called the control store. To execute a machine language instruction, a microprogrammable machine reads a sequence of microwords from the control store and presents them to the control circuits of the computer. Each bit or group of bits in the microword controls an individual data transfer. Thus, one can determine a microprogrammable machine's response to an instruction by coding appropriate microwords into control store. Microprogramming clearly provides a level of flexibility in machine operation that cannot be attained using random logic, by providing control at the level of individual data transfers between elements within the computer.

A manufacturer may choose to implement its machine language on a microprogrammable machine and to provide the user with a fixed set of control store contents (e.g., in ROM). Modifications and enhancements to the control store contents, or microcode, may be possible by changing the control store ROM; but generally, this cannot be done by the user. Such computers are often referred to as microprogrammed machines. Alternatively, a microprogrammable computer can be implemented so that users may write their own data into the control store. These machines are referred to as user microprogrammable, or simply as microprogrammable, computers.

Emulation

Emulation may be defined as the combined hardware/software interpretation of one machine's instruction set by a different machine (Ref 2: 475). The machine being emulated is referred to as the target machine, while the machine upon which the emulation is done is called the host machine. With these preliminary definitions, a statement of the objectives of this investigation is now made.

Statement of the Problem

The overall objectives of this project were:

- To provide sufficient enhancements to MIME and its documentation to allow emulation of target machine instructions by a novice microprogrammer. The desire was that a student with a minimum of training could use MIME as a valuable learning aid in the areas of computer control and microprogramming.
- To provide the Air Force Aeronautical Systems Division Computer Engineering Branch (ASD/ENAIB) with a machine capable of executing the instruction set of MIL-STD-1750, Airborne Computer Instruction Set Architecture (Ref 3). This organization has the primary responsibility

for the standard, and desired a computer upon which to evaluate future changes to the instruction set.

Background

The Microprogrammable Minicomputer Emulator (MIME) is a user microprogrammable general purpose minicomputer. It was designed and built as part of a thesis project at the Air Force Institute of Technology and was intended for use in laboratory investigations of computer control and microprogramming. However, at the end of that investigation (Phase I), the project was incomplete in several areas:

- A complete instruction set had not been implemented on the machine, hence the design had not been verified.
- No software tools for program development, such as any form of translator, were available. Programming MIME was a time consuming and error prone task, since both micro and macroprogramming had to be done in hexadecimal characters.
- All data and program input and output was through a hexadecimal keypad and several readouts on the front panel. Although an RS-232 interface had been designed and built, it had not been fully integrated with MIME (Ref 4).
- There was no microprogramming reference manual generated for MIME.

It was desired to eliminate or minimize the shortcomings of MIME. The actual implementation of a minicomputer instruction set on MIME would be used as the vehicle to pinpoint the specific areas needing improvement. The MIL-STD-1750 instruction set was proposed by ASD/ENAIB.

MIL-STD-1750, Airborne Computer Instruction Set Architecture, established a uniform instruction set for airborne computers to be used in future Air Force avionic weapons systems. The standard was published in February, 1979. However, at the beginning of this investigation no machine existed to execute this instruction set. With this new and untested standard, ASD/ENAIB desired an implementation that would permit modification and experimentation. An implementation by emulation fulfills this desire.

Goals

In consideration of the previous discussion, two goals were set for this project: to emulate the complete instruction set of MIL-STD-1750, and to provide tools and documentation to enhance MIME's usefulness as a pedagogical and research machine. These goals tend to support one another. Realization of the first goal would provide a full minicomputer instruction set for MIME and would help identify the strengths and weaknesses of MIME as a laboratory tool and as a host machine.

Meeting the second goal would not only make realization of the first goal easier, but also would provide future MIME users with sufficient information and tools to allow them to rapidly and correctly microprogram MIME. Information such as user's documentation for MIME, a translator to allow easier programming of MIME, and MIL-STD-1750 emulation documentation would be invaluable in future modifications and evaluations of the MIL-STD-1750 instruction set, in the emulation of other instruction sets, and in the use of MIME as a pedagogical device.

Approach

To reach the goals of this effort, the following steps were taken:

- MIME familiarization
- Related research survey
- Requirements definition
- Hardware design and fabrication
- Software tools design and implementation
- MIL-STD-1750 microcode generation
- Preparation of user documentation

During the first step, familiarity was gained with the MIME architecture, microinstructions, operating characteristics, and existing documentation. This was done by creating small microprograms to exercise various features of MIME. Also during this time many minor errors and inconsistencies in the original MIME implementation were corrected. This entire process was conducted with the assistance of one of the original MIME designers.

The related research survey was conducted to determine what similar investigations were being made in the area of emulation in general and implementations of MIL-STD-1750 in particular. It was hoped if similar work had been done, the results would be applicable to this effort. This would help prevent duplication of previous work and would facilitate reaching the goals of this investigation by allowing the authors to directly apply previous results to the current endeavor.

Requirements definition involved comparing the physical and logical requirements of MIL-STD-1750 with the physical and logical capabilities of MIME. This was done to determine the extent of the changes required to MIME.

Software tools design and implementation embodied the development of a microprogramming language and an associated translator for MIME, as well as the creation of a monitor to allow register and memory location manipulations from a standard RS-232 terminal. These tools were used to generate and test the microcode of the MIL-STD-1750 instruction set emulation.

Preparation of user documentation was an on-going effort throughout the research period. As changes and additions were made to MIME, the existing documentation was updated. As the software tools were developed, the related user documentation was also written.

Related Research Survey

There are several previous and current research efforts related to this investigation. Thourot designed a special purpose microprogrammable machine and the microcode to execute the Software Compatible Avionic Computer Family instruction set (Ref 5). However, this was strictly a paper design. No hardware was built and the design was not verified. The instructions that this computer was designed to execute were in fact a subset and a forerunner of the current MIL-STD-1750 instruction set. The architecture of the machine was quite different from the MIME architecture, and microcode developed for the Thourot machine would have needed major changes to adapt it to execute on MIME.

At least two corporations are working under contract with the Air Force to develop prototype MIL-STD-1750 compatible computers (Ref 6, 7). While the details of either design are considered proprietary information, the contracts call for delivery of the prototype machines, macro/micro cross assemblers, and performance monitoring hardware and/or software. Deliveries are to be made between the last quarter of 1979 and the second quarter of 1980.

Investigation of the two designs showed that each company has developed a special microprogrammable machine to execute the MIL-STD-1750 instruction set. The architectures and microword formats of these machines are significantly different from that of MIME. The micro software and development tools were not available during the research period and would have been difficult to adapt to this project had they been available.

Advanced Micro Devices, Incorporated produces the System 29, a microprogram development system (Ref 8: 3 - 1 to 3 - 10). This package includes a support processor with mass storage and input/output capabilities and software to generate, load, save, and debug microcode. While the System 29 was designed to support the Am2900 series circuits from which MIME was built it has major weaknesses vis-a-vis this investigation.

The microcode generation software (AMDASM) will only support the Am2900 circuits. MIME contains much additional micropro-gram-controlled hardware, for which AMDASM will not generate microcode. Furthermore, AMDASM merely allows the user to define a set of mnemonics corresponding to the commands for each Am2900 circuit. It then processes strings of these mnemonics into microcode. Reading a source program for this assembler is nearly as difficult as reading the microcode it-self since the language cannot express the interactions between individual modules of the computer - this process must be done mentally by the microprogrammer. This concept is contrary to the first objective of this effort. A novice microprogrammer should not be expected to have to learn the circuitry details in order to use MIME.

Organization

Chapter I has provided an introduction to the concepts of microprogramming and emulation, and an introduction to the background and goals of this project. Chapter II provides the comparison between MIL-STD-1750 requirements and MIME capabil-ities, and defines the additional features needed by MIME to perform the MIL-STD-1750 emulation. Chapter III discusses the development of the necessary MIME hardware, while Chapter IV discusses software tools development. In Chapter V, the actual MIL-STD-1750 emulation design is presented, and Chapter VI includes results, conclusions, and recommendations for further investigation. Chapters I through VI comprise Volume I of this report.

There are also several appendices to this document. Appendix A is a MIME User's Manual. Appendix B is a MIL-STD-1750 Emulation User's Manual. Appendices C and D contain reference manuals for the MIME Translator and Monitor, respectively. Appendices A through D are contained in Volume II. Volume III is a large format volume containing:

- Appendix E, MIME Translator Source Code.
- Appendix F, MIL-STD-1750 Emulation Source Code.
- Appendix G, MIL-STD-1750 Mapping PROM Listing.
- Appendix H, MIME Modest Monitor Source Code.
- Appendix I, MIME Schematic Diagrams.
- Appendix J, MIME Parts List.
- Appendix K, MIME Wire-run Lists.
- Appendix L, Program to Generate Wire-run Lists.
- Appendix M, Program to Paginate Pascal Source Listings.
- Appendix N, Programming 2708 EPROM's from Translator Output Data.

Volumes II and III are available from the Electrical Engineering Department, Air Force Institute of Technology (AFIT/ENE), Wright-Patterson AFB, Ohio, 45433.

II. Requirements Definition

This chapter presents a discussion of the methods used to determine the hardware, software, and documentation requirements. The requirements considered were in two broad categories:

- Logical requirements (registers, operations, etc.) of the MIL-STD-1750 instruction set not already implemented on MIME.
- Software tools to help clarify and speed the programming and operation of MIME.

The first category of requirements was determined by comparing, in a systematic manner, the MIL-STD-1750 requirements with MIME's existing capabilities. The following checklist of features was used because it provided a consistent, reasonably comprehensive list of the features of any computer (Ref 9: 38-39):

- Memory
- Registers
- Data
- Instructions
- Special features: Status of a program, input output processing, Interrupts, Masking, Protection, Timers, and Microprogrammability.

Discussions of the MIL-STD-1750 requirements and the Phase I MIME capabilities are presented in the next two sections of this chapter. They are followed by a summary of the comparison. Finally, the requirements for software tools are addressed.

MIL-STD-1750 Requirements

This section presents a brief description of the requirements specified for the prospective target instruction set following the checklist of the previous section. A detailed discussion of each of these requirements may be found in MIL-STD-1750 (Ref 3).

Memory. MIL-STD-1750 requires that the machine by capable of addressing 65,536 words of 16 bit word size. There are various addressing modes required, including register direct and indirect; memory direct, indirect, and indexed; and immediate data.

Registers. Sixteen general purpose 16 bit registers are required, designated R0 to R15. There are also several special purpose registers needed. The Instruction Counter (IC) must be 16 bits in length and external to the general purpose registers. The Status Word (SW) is a 16 bit register whose state is determined by some prior event occurence in the computer. A 16 bit Fault Register (FR) is set by result of various machine faults. The logical OR of the FR bits generates a machine fault interrupt. Formats of the SW and the FR are shown in Figure 1.

Additional required registers are the Interrupt Mask (IM) register and Pending Interrupts (PI) register. These registers allow masking of individual interrupts and a means to

Status Word (SW)

BIT	0	1	2	3	4 – 15
	C	P	Z	N	SPARE
	CS				

Control Status (CS) Bits:

C = 1 iff result generates a carry or borrow.
P = 1 iff result is greater than zero.
Z = 1 iff result is zero.
N = 1 iff result is less than zero.

Fault Register (FR)

BIT	0 1	2 3 4	5 6 7	8 9	10 11 12	13 14 15
	MEMORY PROTECT	PARITY	I/O	ILLEGAL	SPARE	BITE

Bits are defined as follows:

0	CPU is attempting to write into a protected address.
1	DMA is attempting to write into a protected address.
2	Memory parity error.
3	Programmed I/O channel parity error.
4	DMA channel parity error.
5	Output command used with input opcode or input command used with output opcode.
6	Programmed I/O transmission error.
7	Other I/O errors.
8	Illegal address.
9	Illegal opcode.
10-12	Spare for future use.
13	Hardware Built In Test Eqipment (BITE) failure.
14-15	Defined by designer.

Figure 1.

Status Word and Fault Register Formats (Ref 3: 8 - 9) remember which interrupts are still waiting to be processed.

Finally, the multiply and divide instructions of MIL-STD-1750 require up to four registers separate from the general purpose registers. For example, the register transfer description of double precision integer multiply as stated in the standard is

$$(RQ, RQ+1, RQ+2, RQ+3) \leftarrow (RA, RA+1) \times DO \qquad (1)$$

$$(RA, RA+1) \leftarrow (RQ+2, RQ+3) \qquad (2)$$

where

RQ, RQ+1, etc.	are the registers needed to hold the double length product, described in the standard as "logical entities used to clarify the register transfer descriptions".
RA, RA+1	are the two consecutive general purpose registers initially holding the multiplicand and ultimately holding the product.
DO	is the "derived operand", i.e. the multiplier.

The instruction description in the standard specifies that the only registers to be changed are RA and RA+1, implying that while possibly RA and RA+1 could be used as two of the RQ+n, one would still require two additional registers separate from the general purpose registers. The mechanics of register transfers would be simplified if all four RQ+n were separate from the general purpose registers.

<u>Data</u>. The MIL-STD-1750 instruction set is designed to operate on binary data. The data may be single or double precision integers or it may be floating point or extended

precision floating point numbers. All data is stored in its sign plus two's complement form. Figure 2 summarizes the data storage formats.

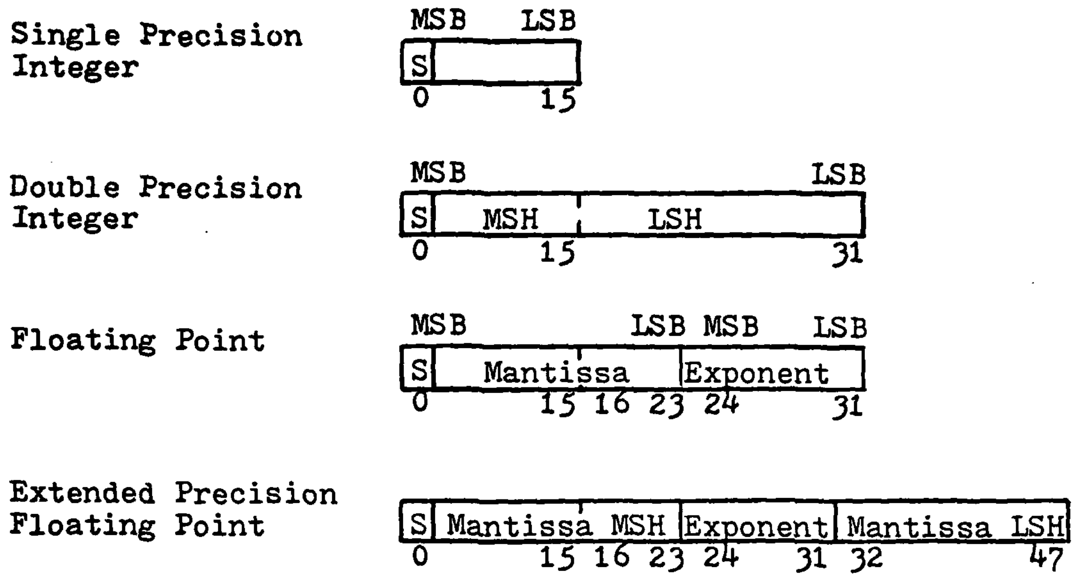
Instructions. Arithmetic, logical, transfer (branching), and I/O instructions are included in MIL-STD-1750. There are also bit manipulation and control instructions provided. A complete list of the instructions and their operation is available in Reference 3.

Special Features. There are several special features required as discussed in the following paragraphs.

Status of a program is kept current by setting the SW, FR, and IM, dependent upon the current state of the machine. The address of the next instruction to be executed by the machine is kept in the IC.

Input/Output (I/O) is handled by a set of special I/O instructions. As with many minicomputers, I/O can be either programmed, interrupt driven or by means of direct memory access (DMA). The I/O instructions are used to transfer information between the general purpose registers and I/O devices or the special purpose registers.

A minimum of 16 interrupts are required as defined in Table I. The interrupts are numbered from 0 to 15 with Interrupt 0 being the highest priority interrupt. All interrupts may be masked or disabled except Interrupt 0 which may not be either masked or disabled, and Interrupt 1 which can be masked but not disabled. Masking causes an interrupt to be held in the Pending Interrupts (PI) register for later



S	=	Sign bit
LSB	=	Least significant bit
LSH	=	Least significant half
MSB	=	Most significant bit
MSH	=	Most significant half

NOTE: All data stored in sign plus two's complement representation.

Figure 2. Data Storage Formats (Ref 3: 2 - 7)

Table I

Interrupt Definitions (Ref 3:10)

Interrupt Number	Interrupt Name
0	Power Down (cannot be masked or disabled)
1	Machine Error (cannot be disabled)
2	Spare
3	Floating Point Overflow
4	Fixed Point Overflow
5	Spare
6	Floating Point Underflow
7	Timer A
8	Spare
9	Timer B
10	Spare
11	Spare
12	Spare
13	Spare
14	Spare
15	Spare
NOTES:	Interrupt Number 0 is highest priority. Priority decreases linearly with increasing interrupt number.

use by the computer.

Each interrupt has associated with it two specified locations in memory to be used as a "linkage pointer" and a "service pointer" for the interrupt. Three consecutive memory locations beginning at the address contained in the linkage pointer are used to store the old IM, SW, and IC contents during an interrupt service. Three other consecutive memory locations, specified by the service pointer, must contain the new IM, SW, and IC contents to be used in servicing the interrupt. Upon return from the interrupt routine, a load status instruction is executed to restore the old IM, SW, and IC values.

Memory parity and memory block protect are optional features. If used, these features set bits in the FR if an error is detected in parity or if a write operation is attempted on a protected memory block.

MIL-STD-1750 requires the implementation of two timers --Timer A and Timer B. Both are count-up, 16-bit timers. Both timers are set, read, started, and stopped using I/O instructions and both generate an interrupt as they "roll over" to zero from maximum count. The only difference between the two timers is that Timer A is incremented each 10 microseconds while Timer B is incremented each 100 microseconds.

There is no stipulation in MIL-STD-1750 whether the computer should be microprogrammed or implemented in random logic. Considering the complexity of some of the instructions (e.g. floating point multiply) and the high probability that some of the machine operations will be modified as the standard evolves, a microprogrammable computer would be a very satisfactory implementation.

<u>MIME Capabilities</u>

This section presents the capabilities of the Phase I MIME compared to the MIL-STD-1750 requirements. A simplified block diagram of the Phase I MIME architecture is depicted in Figure 3. Details of the current MIME architecture can be found in Appendix A, while details of the Am2900 series integrated circuits are available in Reference 8.

<u>Memory</u>. MIME was designed and implemented with a 16-bit Program Counter (PC), corresponding to the IC of MIL-STD-1750 and with a 16-bit word length memory. While the PC allows 65,536 words to be addressed, only 1,024 words of read-write memory have been implemented.

No hardware address decoding is supplied in MIME. The machine instruction is fetched from memory under microprogram control and loaded into the Instruction Register (IR). The contents of the IR may then be interpreted by microcoded routines to determine operand addresses. Use of microprogramming means that MIME can support not only the addressing modes of MIL-STD-1750 but other modes as well.

<u>Registers</u>. The Am2901 4-bit micro processor slice used as the heart of the ALU in MIME contains 16 general purpose registers, each 4 bits in length. Since four of the Am2901 chips are cascaded in the MIME ALU, these registers comprise

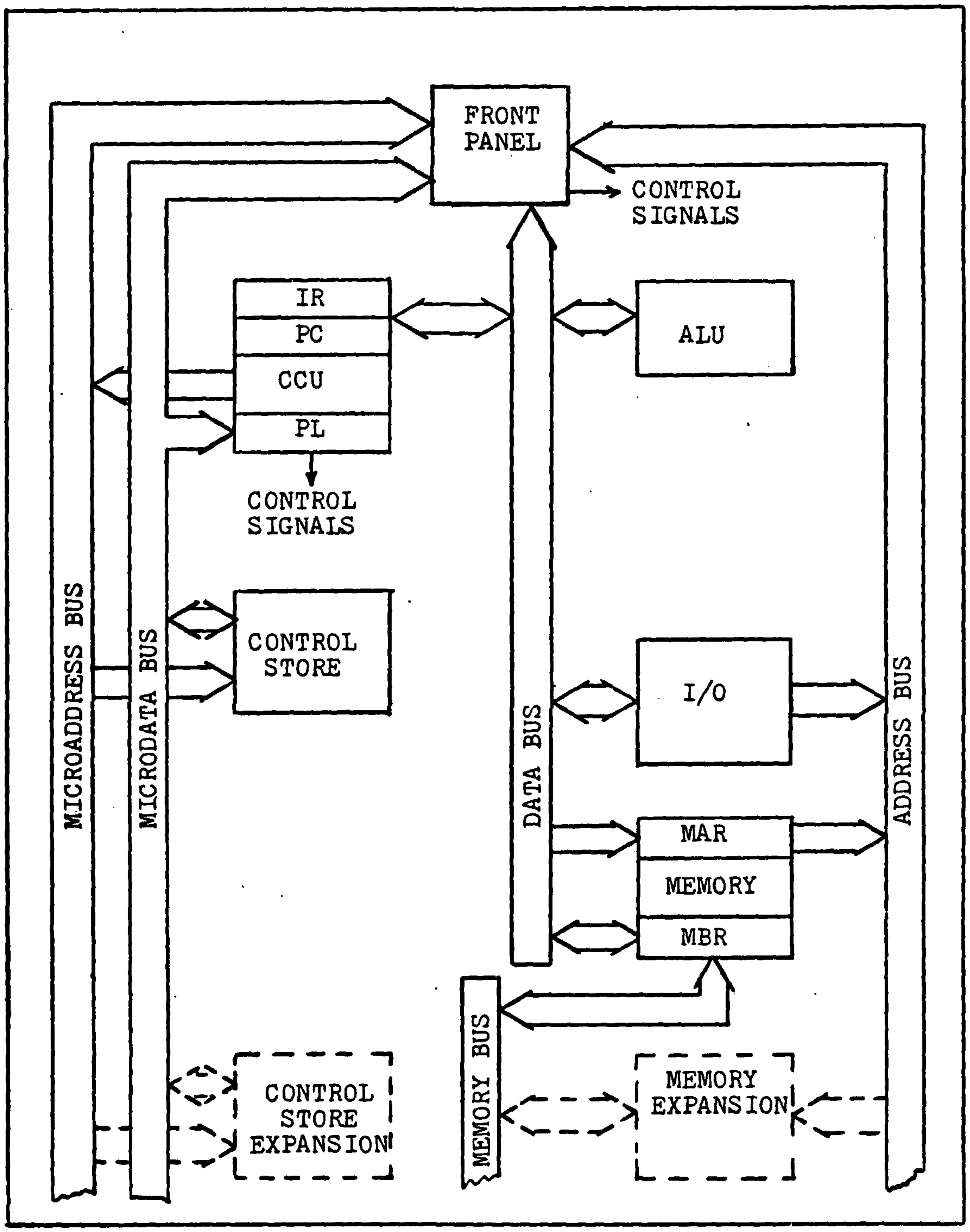


Figure 3. MIME Architecture, Phase I (Ref 1: 81)

the 16 general purpose 16-bit registers needed. These registers can be used to store data used in either microprograms or machine level programs.

The FR required by MIL-STD-1750 was not available, nor was the SW. MIME does have a Condition Code Register (CCR) containing zero, negative, overflow, and carry flags, from which the SW could be generated. The FR and SW functions had to be added to MIME.

The IM and PI registers are implemented in hardware on MIME. These will be discussed under the special features subsection below.

While there is a single Q register on the Am2901 circuit, there was no provision in MIME for the RQ+n of MIL-STD-1750. Consequently, these register functions needed to be added.

Data. There are no hardware restrictions on data types processed by MIME, save that numbers used for calculations in the ALU are expected to be binary. Either sign plus two's complement or one's complement arithmetic can be used.

Instructions. Since instructions are decoded and executed by microcoded routines, nearly any conceivable type of machine instruction can be handled.

Special Features. The special features possessed by MIME that would support MIL-STD-1750 are discussed in the following paragraphs.

Status of a program is monitored in MIME by reference to the CCR. A parallel set of micro condition codes is available to the user to control the status of the micro routine being executed. The PC always contains the address of the next machine instruction to be executed. A micro PC sequences the microprogram.

I/O processing is supported at the microprogram level by MIME. The contents of any register can be transferred onto the data bus and thence to the I/O bus and to peripherals. Similarly, data can be input to any register. "Handshaking" flags are provided for programmed I/O, and all hardware is provided for block transfer direct memory access (DMA).

There were eight levels of priority interrupts implemented on MIME, using the Am2914 Vectored Priority Interrupt Controller chip. This integrated circuit contains all of the registers and priority logic required by MIL-STD-1750. Interrupts may be disabled, enabled or masked by microinstructions, and when an interrupt is requested, the Am2914 outputs an interrupt vector indicating which level of interrupt desires the machine's attention. This vector may be used to branch to a segment of microcode to complete the register manipulations required by the MIL-STD-1750 interrupt scheme. The Am2914 may also be cascaded to provide any number of interrupts.

There is no hardware provision for memory protection or memory parity checking in MIME, although these functions could be microprogrammed. There were no timers implemented on MIME, hence they were outstanding requirements.

MIME is fully user microprogrammable. This means that the user can control each and every transfer of data in MIME by use of appropriate microinstructions. This makes MIME a flexible host machine for emulation of nearly any 16-bit machine's instruction set, provided that the target machine requires a subset of the MIME hardware. MIL-STD-1750 requires a few hardware additions to MIME, as this section has shown.

Comparison Summary

Table II summarizes the comparison between the MIL-STD-1750 requirements and the Phase I MIME capabilities. The requirements column lists the logical requirements to be fulfilled before MIME could be used to emulate the MIL-STD-1750 instruction set. Chapter III discusses the physical realization of hardware to meet the requirements.

Software Tools

This section provides the rationale for the software tools developed during this investigation. Since no software was developed for MIME during Phase I, consideration was given only to the lowest levels of software tools, such as translator or assembler, monitor and loader. Of these, the tools judged most crucial were a translator and a monitor. These are discussed in the following paragraphs.

Translator. At the end of Phase I, the MIME hardware was essentially complete but no software development had been done. The generation and testing of a complete instruction set emulation microprogram would have been exceptionally difficult if the entire program had to be hand coded and hand entered into MIME. Therefore, a requirement was identi-

Table II

MIME/MIL-STD-1750 Comparison

Feature	MIL-STD-1750 Requirement	MIME Capability	Hardware Requirement	Remarks
Memory	64 K words x 16 bit max.	64 K words x 16 bit possible	None	1 K words memory implemented.
	7 addressing modes	addressing is microprogrammed		1 K control store implemented
Registers	16 gen. purp. Instruction Counter (IC)	16 gen. purp. Program Counter (PC)	None None	
	Status Word (SW)	Condition Code Register (CCR)	SW	SW bits derived from CCR.
	Fault Register (FR)	None	FR	
	Interrupt Mask (IM)	Mask Register (MR)	None	Am2914
	Pending Interrupts (PI)	Interrupt Latches	None	Am2914
	Auxillary Regs. (RQ+n)	1 Q register	RQ+n	Used for mult. and divide.
Data	16 bit words integer and floating point	16 bit words	None	
Instructions	Arithmetic Logical Transfers I/O	All types microprogrammed	None	
Special Features				
I/O	Between gen. purp. regs. and devices	Between any register and devices	None	
Interrupts	16 levels	8 levels	Expand to 16	
Masking	Masked interrupts held pending	Masked interrupts held pending	None	Software setable
Protection	Optional	None	None	
Timers	Timers A and B	None	Timers A & B	
Microprogrammed	Not specified	Yes	None	

fied for a translator to convert programs expressed in mnemonics or in English words into MIME microcode. Such a translator would be the micro level counterpart of an assembler.

Monitor. The second major software tool required was a monitor which would allow a user to examine and load registers and memory from a terminal. Although the MIME front panel allows these functions, MIME must be halted to examine or load most registers. Furthermore, the existing front panel would not allow loading or displaying of the new registers required by MIL-STD-1750. The monitor should allow the user access to all registers of interest and should allow a user to suspend a program in execution, examine and/or change registers as desired and return to the execution of the program. By having these capabilities, the monitor would be a useful tool in the execution and debugging of both micro and macro level programs.

Summary

This chapter has presented a discussion of the hardware and software requirements for Phase II of the MIME project. Chapter III will present a discussion of the realization of the hardware requirements, while software tool development is the subject of Chapter IV.

III. Hardware Realization

This section presents a brief discussion of the alternatives considered and the solutions chosen to meet the hardware requirements determined in Chapter II. It should be noted at this point that, at the microprogramming level, the distinction between hardware and software implementations of the same function is less clear than at the higher levels of machine programming. Most changes or additions made to the microprogrammed machine require changes to both the hardware and the microinstruction format and function. Consequently, in this chapter, after the hardware changes are discussed, the microinstruction changes needed to make the new hardware work will also be presented; and in turn, the required microword decoding hardware changes will be covered.

Auxiliary Registers

The Problem. As stated in Chapter II, the auxiliary registers, RQ through RQ+n, are only needed in multiplication and division to hold intermediate results of the calculation, and as work space for the operation. To understand the required operations of these registers one must understand the algorithms used for multiplication and division. Multiplication is discussed here although the considerations given apply equally to division, which may be viewed as "multiplication in reverse". A simple algorithm for integer binary multiplication by repeated addition is given below (Ref 2: 53):

1. Start with accumulated product equal to zero.
2. Inspect the multiplier bits individually, starting with the least significant bit.
3. Add the multiplicand to the accumulated product if the multiplier bit is a 1; otherwise add 0's.
4. Shift the accumulated product and multiplier one bit to the right.
5. Go to step 2 and repeat the loop until all the partial products are added.

While many faster and more elegant algorithms exist (Ref 2: 54-57, 10: 129-156), this algorithm is the most straightforward and provides the greatest insight into the actual machine operation.

There are several requirements dictated by this algorithm. First, the register which is to hold the product must be clearable. This is possible either by loading the register with zero or by using a direct clear function. Secondly, the register holding the multiplier must provide the capability of having its bits examined one at a time. An easy implementation of this operation is to test the least significant bit followed by performing a one-bit right shift before testing the next shift. Finally, the register holding the multiplicand needs a left shift capability.

Possible Solutions. The first solution considered was to dedicate part of the 16 registers in the ALU to this task. The contents of the registers, which would likely be values in use in the machine program, would be copied into an external memory area, either in main memory or in a dedicated register file. At the end of the instruction execution the contents of the registers would be restored to their correct values. This approach had many disadvantages, chief among which were:

- Machine registers other than those specified by a given machine instruction may be altered in the execution of that instruction. If the microprogram were halted before the execution of the machine instruction is complete, as in a machine error condition, the contents of the registers would not be those expected by the machine programmer with no knowledge of the microroutines involved. This would make debugging a very difficult task, as well as being a contradiction to the MIME design philosophy of making both micro and macro operations as visible to the user as possible.
- The add operation of step 3 and the shift operation of step 4 of the algorithm require one pass through the ALU for each 16 bits being processed. This also would require frequent recomputation of the A and/or B latch addresses for multiple precision operations.
- Using main memory to store the information was not desirable since this scheme would reduce the generality of the MIL-STD-1750 emulation. Mixing of user memory with memory needed by the microcode would mean that the microroutines would not be transparent to the machine level programmer and that the full 64 K of user memory could not be available for user programs.

The next implementation considered was that of simply using the external memory area as the work and intermediate storage area. This scheme eliminated the problem of disturbing the general purpose registers, but the second and third disadvantages above remained. Also, each ALU operation would require the operands to be fetched into the DBR before they were used and subsequently restored via the DBR. This would double the shift and add times, and these two operations are repeated many times in a multiplication.

The final alternative considered was to use external hardware shift registers designated AUX1 through AUX6. These registers could be manipulated as shown in Figure 4. This scheme eliminated the frequent recalculation of operand locations and avoided unnecessary alteration of the general purpose registers.

Implementation. The last alternative of the three above was deemed the most feasible. It eliminated interference with user memory and allowed simpler code, since latches and registers need not be "juggled". The registers were constructed using the 74S299 universal shift register. This integrated circuit provides an eight bit register with bidirectional data lines, left and right shift capabilities, and tri-state outputs in a single 20-pin package. Two 74S299's were used for each AUX register and all were connected to communicate directly with the MIME data bus. Details of the circuit and its operation are provided in Appendix B.

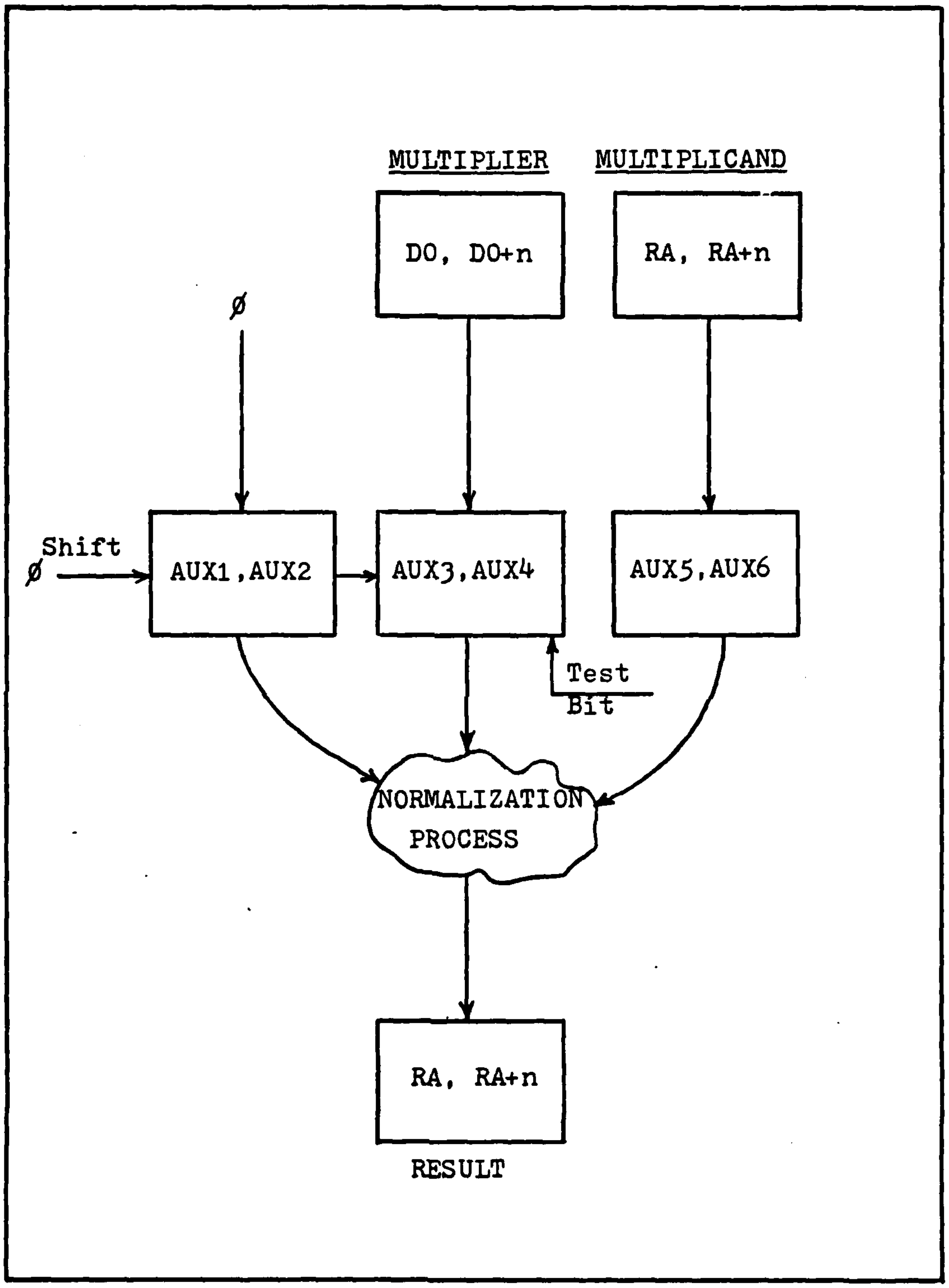


Figure 4. Use of the Auxiliary Registers

Status Word (SW)

The Problem. Although MIME possessed a condition code register (CCR), it contained only bits indicating carry, zero result, negative result, and overflow as possible results of ALU functions. The MIL-STD-1750 SW needs the carry, zero, and negative bits, but also needs a positive result bit. Additionally, the standard requires the availability of twelve more bits to be defined at a later date.

Clearly, the four bits presently required for the SW could be generated from the CCR flags. The positive result bit is simply the logical NOR of the zero and negative bits. The problem was how to physically implement this register.

Possible Solutions. One possible solution considered was to use a memory location to hold the SW information. This was deemed undesirable for two of the same reasons stated in the discussion of the auxiliary registers: loss of generality, and additional use of ALU required to update the register contents. This scheme was abandoned in favor of a special purpose hardware register.

Implementation. The required status bits for the SW were generated as indicated above. These bits were then multiplexed with the data bus into the inputs of a 74374 register. Clocking was applied to the register such that each time the CCR was loaded in MIME, the appropriate bits would be loaded into the SW register. The register was connected to be loaded from or read to the data bus, so that its contents could be saved and restored during interrupt handling, and so that the status could be read and used to determine the results of conditional branching instructions. Full details of the circuit are presented in Appendix B.

Fault Register (FR)

The Problem. The FR is read and cleared by use of I/O instructions. Furthermore, anytime one or more of the FR bits are set, a machine error interrupt must be generated.

Implementation. Although the possibility of using a memory location for the FR was considered, it was rejected in favor of a hardware register. The registers used were the Am2918 quad register. This integrated circuit has two outputs for each bit of the register; one is a tri-state output, while the other is a standard totem pole output. Four registers were connected to be loaded from or read to the data bus via the tri-state outputs. The standard outputs of all bits were OR'ed, with the result providing the needed machine error interrupt request.

Interrupts

The MIME was originally provided with eight levels of priority interrupt implemented by the Am2914 Priority Vectored Interrupt Controller. As previously stated, this integrated circuit is very easy to cascade to give any number of interrupt levels. Thus, the only implementation considered was to add another Am2914 to MIME, yielding the required sixteen levels of interrupts. Circuit details are in Appendix A.

Timers

The Problem. Timers A and B are required to be 16 bit count-up timers. Each timer must be capable of being loaded, read, started and stopped by means of I/O instructions. When either timer rolls over from maximum count to zero, an appropriate interrupt must be generated. MIME contained no capabilities such as these.

Possible Solutions. The timers could have been implemented using a single integrated circuit such as the Intel 8253, which contains three counters that can be used as timers (Ref 11: 10-159 to 10-169). However, these circuits are typically designed to interface with an eight-bit microcomputer and hence have only eight I/O lines although the internal counters are sixteen-bit counters. Thus, the data from MIME would have to be latched into registers and transferred eight bits at a time for a load operation and vice versa for a data transfer from the timers to the MIME bus. The circuitry required for the latching and to decode the MIME control signals into appropriate controls for the 8253 would have increased the chip count enough to overwhelm the advantage of having the counters on one chip.

The other solution, which was adopted, used four 4-bit binary counters for each timer. This solution provided a straightforward implementation and made use of readily available standard integrated circuits.

Implementation. Each timer consists of four 74193 counters with the outputs driving the data bus through tri-state buffers. The timers can be individually loaded from the data bus. A 74124 dual voltage controlled oscillator circuit provides the clocks for the timers with one half running at 100 KHz and the other at 10 KHz. The zero count of each timer is decoded to provide the appropriate interrupt request. Details of the circuit may be found in Appendix B.

Control Store Expansion

The MIME was originally built with 1 K words of microprogram control store, but the addressing capability and necessary decoding were provided to expand the control store to a maximum of 4 K words. It was decided to expand the control store to its maximum size by adding the extra 3 K words on a separate printed circuit board. This way the lowest 1 K words of control store might be used to store resident MIME utilities, such as a monitor, while the higher addresses in control store might be used to hold the microcode necessary for a particular target machine emulation. Changing the second control store board would then be a rapid way to change the MIME's characteristics from one target machine to another.

The second control store board, designated CS2, was implemented on an Augat 8136-URG5 card to be compatible with the other MIME modules. Type 2708 erasable programmable read-only memories (EPROM's) were used primarily because the 2708 is both readily available and programmable. Details of the CS2 circuitry are provided in Appendix A.

Microword Changes

Some changes had to be made in the microword format to control the added registers. The required control signals were defined to be:

- Read registers/timers (individually)
- Load registers/timers (individually)
- Start/stop timers
- Clear/shift AUX registers

The load and read operations were accomodated using spare codes in the BUS DESTINATION (bits 4 - 7) and BUS SOURCE (bits 8 - 11) fields of the microword, but some means had to be found to select the individual register being read.

As mentioned in the Introduction, early in the investigation, several errors and incongruities in MIME's design were eliminated. In doing this, the functions of two fields in the microword, the ALU A ADDR (bits 20 - 23) and the ALU B ADDR (bits 16 - 19) were eliminated. These fields were utilized to control the new hardware. Bits 20 - 23 were newly defined as auxiliary command (AUX COMMAND) and were activated by a spare code in the COMMAND field. The new commands were named User Definable Auxiliary Commands (UDAC, followed by a hexadecimal number). There are 16 possible UDAC's (UDAC0 - UDACF).

Similarly, bits 16 - 19 of the microword were designated the auxiliary function (AUX FUNCTION) field. These microword options were referred to as User Definable Auxiliary Functions (UDAF, followed by a hexadecimal number). There are 16 possible UDAF's (UDAF0 - UDAFF).

By selecting the Aux Load Code (D) in the BUS DESTINATION field along with a properly defined AUX FUNCTION code, any register or timer can be loaded. Similarly, to read a register the Aux Read code (5) is selected in the BUS SOURCE field along with the appropriate AUX FUNCTION code. The AUX COMMAND codes accomplish the miscellaneous command functions required to control the new registers and timers.

Interface to MIME

Interfacing the new hardware needed to emulate the MIL-STD-1750 computer presented the problem of maintaining MIME's generality. Any permanent changes to MIME hardware must not be peculiar to any target machine, yet some target machines might need peculiar hardware, as in the case of MIL-STD-1750.

This problem was circumvented by dividing the new hardware into two groups: hardware generally applicable to MIME operation, and hardware peculiar to MIL-STD-1750. The latter included the AUX registers, the SW and FR, and the timers. These were implemented on an AUGAT 8136-URG5 circuit card, with all control signals and test conditions as well as the data bus brought out to the edge connector. This card was designated AUX, and scheduled for installation at station I of the MIME card cage.

The permanent MIME hardware additions consisted only of expansion of the number of interrupts using an added Am2914 and hardware to decode the AUX COMMAND and AUX FUNCTION fields of the microword. These decoded signals were made available at the edge connector at station I in the MIME card cage.

The net result of this interface scheme was that a user could easily reconfigure the MIME hardware to a specific application by installing a new AUX card and redefining the AUX COMMAND and AUX FUNCTION fields in the microword. This action, coupled with the development of suitable microroutines, would allow the user the flexibility to configure MIME to meet the requirements of many diverse applications.

For example, in this effort the AUX card was designed to implement registers and timers peculiar to MIL-STD-1750. The microroutines were developed to execute the MIL-STD-1750 instructions, and the microcode was stored on CS2. A future user may decide to emulate, say, the XYZ machine, a mythical machine with an architecture different from that of MIL-STD-1750. The user could remove the MIL-STD-1750 AUX board, and replace it with a new board designed to fit the XYZ architectural requirements. Then, by replacing the MIL-STD-1750 microcode (either by replacing or reprogramming the PROM's on CS2) with appropriate routines to execute XYZ instructions, the user will have transformed MIME into an XYZ machine emulator.

Summary

This chapter has presented the development of required hardware additions to MIME to allow emulation of the MIL-STD-1750 computer. The rationale for the design decisions and hardware selections was given as well as a discussion of the results of those actions.

IV. Software Tools

Introduction

This chapter describes two software tools that were developed to facilitate other tasks involved in this investigation. The MIME language and the translator consist of a language used to mnemonically describe the microprogramming operations and a Pascal program that translates the language into the equivalent microcode. The MIME Modest Monitor (MIME/MM) consists of a microprogram that communicates through a terminal and allows the user to load and examine memory and register values. It was specifically designed to aid in the debugging phase of the emulation, but would be useful to any microprogrammer.

MIME Language and Translator

Goals. The MIME language and translator were designed towards three primary goals:

- A translator program was highly desirable to ease the burden of microprogramming the MIME. Converting a software design into the microcode for MIME's 64-bit microword by hand was a tedious and error-prone process.
- A method was needed of insuring that errors in the microcode were kept to a minimum. By eliminating the need to hand-manipulate the microcode, the translator represented a large step in this direction. It was

also feasible to build into the translator other checks with which the microprogrammer would otherwise have to be concerned. This included such items as which operations can be performed concurrently, the specifics of setting up ALU operations, and detecting operations that would produce contention for the data bus.

- It was desired to improve the readability and understandability of a microprogram. MIME microcode, in a 64-bit or 16-hexadecimal-characters object form, is unintelligible without extensive familiarity with the format and fields of the microword. Adding comments to this object format, in the form of a verbal description of the microinstruction or its register transfer language equivalent, aids in the understanding but also dictates that additional, unusable (to the machine) information must be carried with the object code. By mnemonically describing the operations of microprogramming and letting a translator program be concerned with the object code, these problems assumed a lesser significance.

These goals all stem from one of the two goals of this investigation, namely to enhance MIME as a pedagogical tool. The translator allows a novice microprogrammer, as would be found in a classroom environment, to begin microprogramming more readily. It relieves him of the burden of becoming an expert in the architecture of MIME and its microword before being able to do even the most basic of tasks. Although the translator will not allow every possible option in the microcode (certain possible concurrent operations cannot be done in the translator), the microprogrammer is assured that any microprogram that translates successfully will perform the operations on MIME described in the program. Programs that are not written properly will produce suitable error messages for the programmer.

Language Rationale. The MIME language was generally designed to follow the machine-level operations available in the microword. In most cases, high level language-like structures and mnemonics, rather than the more cryptic assembly language style, were used to express the operations. This resulted in a more readable and understandable program. The language maintained the assembly language feature of a one-to-one correspondence between the microinstruction and the resulting microcode. Also incorporated were various assembler directives to allow the microprogrammer options on output formats and control of the microaddress.

There are a large number of reserved words due to the restrictions on the operations that are capable of being expressed by the various fields of the microword. In order to avoid unnecessarily restricting the microprogrammer, a feature was included in the language to allow a redefinition of most of the reserved words. This allows the programmer to substitute another set of mnemonics which may have more meaning to the particular problem addressed by the program.

Translator Rationale. Consideration was given to available programs that would perform the translation. For example, Advanced Micro Devices produces AMDASM, a "meta-assembler" that will translate mnemonics into microcode (Ref 8: 4 - 1). The primary drawback of this system was that it basically allows only substituting a mnemonic for a value to be inserted into a field in the microword. This produces a source program of single word mnemonics and does not permit adding connective words to give the language a free flowing format. It was felt that the readability of the langauge to the non-expert microprogrammer was an important enough consideration to discard AMDASM and other similar translator programs in favor of one designed specifically for MIME and its environment.

What has been referred to as a "translator" throughout this report is in reality a cross-translator, since it runs on another machine. Writing a true translator to run on MIME and translate the MIME language into MIME microcode was deemed inappropriate since control store, where the translator program would be resident, is a very limited resource on MIME, being limited to 4K microwords by the microaddressing capability of the machine. It was also an informal objective to avoid extensive microprogramming by hand.

The translator, being specifically designed for the MIME microword, will probably not see significant changes in the future. However, features were built into the translator that aided in the initial design and would aid in future modifications. The translator will operate in a debugging mode, where various informative messages are printed as the translator processes each microstatement. A consistency summary of reserved words and their use, along with a summary of the various microword functions and what reserved words are related to the functions, can also be printed to help in changes to the reserved words and MIME function definitions. These features are triggered by pseudo-operations in the translator. They are not explained in the translator reference manual (Appendix C) to hide them from the average microprogrammer. However, a maintenance programmer will find them described in the translator source code comments (Appendix E).

A post-processing phase was included in the translator to prepare the microcode produced in the translation for actual use. This produces an object file that is either directly loadable into MIME or which can be used to program PROM's (both with suitable interfaces). A technique for generating mapping PROM's is also included, since a major use of microprogramming is to emulate another instruction set.

The translator was designed to provide the programmer with thorough error checking. In addition to syntax errors for the language, this also includes checking for multiple use of a microword field, contention for use of the data bus, attempts to perform operations concurrently that cannot be done concurrently, and complete specification of functions (e.g., add operations must have a carry-in bit). Label and branching usage and consistency are also examined.

Language Structure

This section describes the MIME language structure. An overall description of a microprogram is provided, followed by a brief discussion of the translator input and output files.

The language was designed around the capabilities of the MIME microword, which in turn reflects the features of the Am2900 series of integrated circuits used in the fabrication of MIME. The various fields in the microword are combined to accomplish the functions defined by the language.

Microprograms. A microprogram consists of a redefinition section and a program section, separated by the word PROGRAM, and terminated by the word END. The redefinition section allows the programmer to redefine most mnemonics of the language in terms directly suitable to the problem being programmed. For example, if a machine is being emulated that uses an instruction counter (IC) rather than a program counter (PC) as found in MIME, the programmer may consider redefining those words associated with the PC. The redefinition feature also allows the programmer to use more suitable names for the auxiliary functions and commands.

The program section consists of microstatements from the language. Comments, denoted by the "#" character at their beginning and end, are allowed at the end of a microinstruction. Labels can precede any microinstruction and are identified by their terminating colon (e.g., LABEL1:). The colon may be followed by a label type, which is necessary for labels used in multi-way branches and looping statements, as explained later. Constants are denoted by apostrophes, such as 'ØØ1F'. Pseudo-operations are available to direct the operation of the translator and the format of the output files.

Input and Output Files. MIME language microinstructions are input to the translator on a free-format, 80 column card image file. More than one microstatement may be permitted in a single microword, depending on the operations that can be performed concurrently and which fields of the microword are used. Each microstatement is terminated by a semicolon, and a series of microstatements that form a microinstruction is terminated by a period.

Output from the translator consists of a listing and an optional object code file. The listing consists of the source microstatements, the microcode generated, the microaddress for that microstatement, plus error messages and summary information regarding the translation process. This can be used to input the microcode into the MIME using the front panel keyboard. Alternatively, post-processing of the translator output can be invoked using a pseudo-operation. This causes the microcode to be written to the object file after additional processing. Suitable software (e.g., see Appendix N) can then use the object file to program PROM's.

When the translator encounters an error in the input microstatements, an error message is printed, along with a pointer to the approximate location where the error occurred. The translator attempts to recover from the error and continue translating as gracefully as possible. As a last resort, it will scan ahead until it finds the end of a microstatement (a semicolon) and resume translating at the beginning of the next microstatement.

Microstatements

The microstatements of the MIME language can be divided into eight functional groups. Explanations of the specific effects of each microstatement can be found in Appendix C. In this section, an overview of the features of these functional groups of microstatements will be presented. A complete Backus-Naur Form (BNF) definition of the language, along with an explanation of the BNF notation, can be found in Attachment D of Appendix C.

Bus Transfer Microstatements. This group of microstatements governs the movement of information over the data bus between the functional units and registers of the MIME. The format of the microstatement is

LOAD <bus destination> FROM <bus source>

Information transfers on the data bus are usually determined by the BUS SOURCE and BUS DESTINATION fields of the microword, although other fields may also influence the information flow. For consistency, all bus transfers use the same format, regardless of which microword fields are affected. Memory transfers also use this form of the microstatement, where the word MEMORY is used as either the bus source or destination. The User Definable Auxiliary Functions (UDAF's) are also considered bus transfer operands.

ALU Function Microstatements. This group of microstatements governs the operation of the ALU. These microstatements mnemonically describe the operation of the Am2901 ALU (Ref 8: 2 - 3) and follow the syntax

<ALU operand> <ALU operation> <ALU operand> =<ALU destination>

Only certain operand pairs are acceptable to the ALU. However, in many cases, the ALU operation is such that the order of the operands is irrelevant (e.g., logical OR). In these cases, the translator determines the appropriate operand pair. In cases where order of operands is important, the translator only allows the operands in the correct order. Since there are two versions of the arithmetic subtract operation, the translator selects the appropriate one based on the order of the operands. The only invalid operand pairs, regardless of order, are where both operands are the same (e.g., A XOR A), the pair DBR and B, and the pair Q and B.

The arithmetic add and subtract ALU operation both require that a carry-in bit be specified. This is appended to the plus or minus sign, as in the microstatement

A +:1 B = BF

signifying that the carry-in for this add operation is a "1".

The naming convention for the ALU destination mnemonic was developed to allow easy identification of the destination. The first two characters may show that the result is shifted before being stored. The next characters show the location where the result is stored. The last character shows what is available on the data bus from the ALU. As an example, an ALU destination of LSBF indicates that the result of the ALU operation (the "F" output of the ALU) is available to the data bus and is stored, left shifted by one bit, into the register addressed by the B address latch.

Some of the destinations of the ALU function signify that the result is to be shifted left or right one bit before being stored in the B register or the Q register. In these cases, the register addressed by the B latch is viewed as two eight-bit bytes and the Q register as one sixteen-bit word. Therefore, the programmer must specify what is to be used as the source for the bit shifted into the Q and B registers and what is to be shifted between the two bytes of the B register. This information is required to be appended to any shifted destination field of the ALU function.

Disable Byte Microstatement. These microstatements are used to disable either the high order eight-bit byte or the low order eight-bit byte when storing the result of the ALU operation. These microstatements take the form of either DISABLEUP or DISABLELOW, with the default (when neither is specified) of both bytes being enabled.

Set Condition Code Microstatement. These microstatements are used to specify which set of condition code flags to load and the source of the data. Macro or micro condition codes are selected using SETCC or SETMCC, respectively. Since there are four condition codes, up to four sources for the setting of these flags may be specified, with the default for unspecified codes being to set that code based on the corresponding current macro condition code. For example, the microstatement

SETMCC : BNEG, WZERO, CARRYØ

would cause the micro negative flag to be set based on whether the selected byte was negative, the micro zero flag to be set if the data word is zero, the micro carry flag to be cleared, and the micro overflow flag, since it was not specified, to be set based on the macro overflow flag.

Interrupt Control Microstatements. These microstatements affect the interrupt control field in the microword and correspond to the commands acceptable to the Am2914 Vectored Priority Interrupt Controller (Ref 8: 2 - 134). Each of these microstatements corresponds to a setting of the I/O field in the microword.

Command Microstatements. These single-word microstatements are used to describe predefined actions and data transfers available in the MIME. Such actions as INCPC (increment the PC), HALT (stop executing microinstructions), IOWRITE (write to an I/O device), and DECB (decrement the B address latch) fall into this category of microinstructions. Each of these microstatements corresponds to a setting of the COMMAND field in the microword.

Auxiliary Command Microstatements. These microstatements set the user definable auxiliary command field of the microword. This field is available to an auxiliary circuit board, which determines the effect of these microstatements.

Transfer Microstatements. This group of microstatements is based on the capabilities of the Am2909/2911 Microprogram Sequencers, the Am29803 16-Way Branch Control Unit, and the Am29811 Next Address Control Unit (Ref. 8:2 - 85). These devices interact to produce three types of transfer microstatements, consisting of the multi-way branch, the unconditional transfer microstatement, and the conditional transfer microstatement.

The multi-way branch microstatement allows the programmer to transfer control to one of up to 16 mictostatements. In the microstatement

ON IRØ, IR1, IR2 GO TO DECODER

the label "DECODER" specifies the first microstatement in the list of possibilities to select from. The branch control field (IRØ, IR1, IR2) determines which microstatement in the list to transfer to. In actuality, the hardware logical OR's the three bits in the branch control (bits Ø, 1 and 2 - the three least significant - of the instruction register) with the least significant three bits of the microaddress of the label "DECODER", yielding eight possible branch destinations in this case. The label "DECODER" should be the first of eight possible microstatements from which to select. One to four bits can be specified in the branch control field of the microstatement, yielding two, four, eight and sixteen way branches. The translator places the start of the list of branch destinations on a two-, four-, eight- or sixteen-word boundary - as specified by the label type when the label is defined - so that the OR operation will work properly. For example, the label "DECODER" should have been defined as

DECODER:8

since it is being used for an eight-way branch. A higher numbered label type (e.g.,16) would also work, since a sixteen-way branch label would also be on an eight-word boundary.

Unconditional transfer microstatements reflect all the possibilities that can be generated by the Am29811 Next Address Control Unit if the condition is forced to either true or false. These consist of branch microstatements (GO TO), subroutine microstatements (CALL, RETURN), stack operations (PUSH, POP), and loop counter operations (LOADCNTR, LOOP AT). For example, the destination of a GO TO microstatement can be either a label, REGISTER (which always contains the branch address field of the previous microinstruction), START (a switch-settable address that is transferred to on a front-panel reset), ZERO (microaddress zero), and MAP (the output of the mapping PROM). The loop counter load instructions are consistent with the format for the bus transfer microstatements, since the loop counter is loaded from the data bus. For example, the microstatement

LOADCNTR FROM ALU

would gate the output of the ALU on the data bus and load the loop counter with the least significant eight bits of this data. The load counter microstatement could have been a part of the set of bus transfer microstatements, but since the counter is controlled by the microaddress sequencers and because of its capability of being used on a conditional microstatement, it was made a transfer microstatement.

The conditional microstatement follows either an IF-THEN or an IF-THEN-ELSE format. The condition is selected from the macro or micro condition codes or various other status bits available throughout MIME. The condition may be preceded by the word NOT, causing the condition to be negated. The transfer microstatements acceptable as options to be executed on a true or false condition are a subset of the unconditional transfer microstatements. The Am29811 Next Address Control Unit has sixteen commands, each of which can select one of two unconditional transfer microstatements depending on whether the condition is true or false. Therefore, the IF-THEN-ELSE construct is restricted to one of those sixteen pairs of microstatements (there are only five useful and valid pairs) and the IF-THEN construct is limited to the pairs that have a CONTINUE microstatement as one of the pair (there are six of these). Examples of the conditional microstatement are:

```
IF NEG THEN CALL FIXNEG

IF DMAOVER THEN GO TO DMAENDING ELSE GO TO REGISTER
```

<u>MIME Modest Monitor</u> (<u>MIME/MM</u>)

This section provides the goals and rationale for the design of a minimum-capability interactive monitor. In view of the modest capabilities of this monitor it was named the MIME Modest Monitor (MIME/MM).

Goal. The goal of the MIME/MM development was to provide a convenient means of loading and examining MIME's registers and memory. Although the registers and memory locations could be manipulated individually from the front panel, using the front panel was neither rapid nor convenient. Furthermore, when the user definable functions were implemented as registers, these registers could not be either loaded or examined from the front panel.

Rationale. What was needed was a monitor program to allow register and memory manipulations from a user terminal or through an external computer acting as a controller. This monitor had to be microprogrammed, since no machine instruction set was yet implemented. Also, a microprogrammed monitor could remain resident in MIME to be used in conjunction with any emulation.

The overall design strategy of the monitor involved taking a "snapshot" of the MIME's registers at the time of MIME/MM invocation by stacking the contents of the registers in predetermined reserved memory locations. The user could then examine the "register contents" by examining the stack of values in memory using MIME/MM commands. Should the user desire to change the contents of a register, the appropriate memory location would be changed. The user could then exit from the monitor by invoking an exit routine to unstack the register values and to restore the register contents. This approach was used because several registers were needed to run the monitor program itself and their contents would have thus been altered. Memory changes could be done directly as they were commanded by the user since, excepting the stack, memory was not affected by the MIME/MM.

One additional feature was added to the MIME/MM - the ability to dump consecutive locations of memory to the terminal for storage on paper or cassette tape, and to reload those tapes. This would allow the user to save and reload machine level programs, as well as to load the machine code generated by an assembler for a target machine (provided, of course, that this code was in the correct format).

The functions mentioned above - dumping and loading registers, dumping and loading memory to a terminal for display or storage and exiting the monitor to the user program - were deemed sufficient to perform the monitor's function when used with a terminal. These basic functions could be modified to suit the user by interfacing a mini or microcomputer to the MIME I/O port and interacting with MIME/MM from expanded monitor program running on the external computer. A discussion of this possibility is beyond the scope of the current investigation, however, and is left for the Recommendations section of Chapter VI. A description of the MIME/MM program and its operation is available in Volume II, Appendix D, MIME Modest Monitor MIME/MM User's Manual.

MIME/MM Functions

This section provides an overview of the MIME/MM capabilities.

Dump Registers (DR). This command dumps the contents of the registers at the time MIME/MM was invoked. The data is read from the memory stack, formatted and sent to the terminal.

Load Registers (LR). The operator is prompted for a register name and a value to put in that register. The value is stored in the memory stack until the registers are reloaded upon exit from MIME/MM.

Load Memory (LM). The operator is prompted for the address at which loading is to begin. After receiving the address, MIME/MM waits for the user to input data for as many locations as desired. The data is actually loaded into the memory locations as it is entered.

Dump Memory (DM). This command works like LM, except that the contents of consecutive memory locations are sent to the terminal.

Dump to Tape (DT). The operator must input the start address and the number of words to be dumped. The data is sent from memory to the terminal for recording on magnetic or paper tape.

Load from Tape (LT). This command allows loading the tapes made by using DT. The operator must input the start address and number of words to be loaded into memory.

Exit (E). This command causes the register values to be unstacked and loaded into the MIME registers. Control then passes to the microaddress specified as IFTCH in the MIME/MM code. This command was used by the authors to branch to the machine level instruction fetch routine.

Initialize (I). This command performs the same as E, except that it passes control to a different location. This command was used to branch to the machine initialization microroutine.

Summary

This chapter has presented the design goals and rationale for two software tools for the MIME - the MIME translator and the MIME Modest Monitor. These tools were designed to have general application to any microprogramming effort and to ease the task of programming, testing, and debugging microcode.

The previous chapter discussed the hardware changes made in the MIME to emulate the MIL-STD-1750 instruction set. This chapter has provided a discussion of the development of the software tools generated to make the emulation a tenable task. In the next chapter, the actual MIL-STD-1750 emulation design is covered.

V. MIL-STD-1750 Emulation

Introduction

This chapter provides an overview of the hardware and software configuration that was created to emulate the MIL-STD-1750 instruction set. The hardware development was based on the comparison of requirements and present MIME capabilities as defined in Chapters II and III. The software (microcode) development was based on the register transfer language and associated explanations contained in MIL-STD-1750 for each of its instructions.

Deviations from MIL-STD-1750

In some cases, there are differences between the emulation described in this report and MIL-STD-1750. These occurred as a result of:

- Options available in the description of the instruction set.
- Ambiguity in the description of the instruction set. Since MIL-STD-1750 is relatively new, these discrepancies might be expected. When possible, consultation was made with ASD/ENAIB (the organization responsible for the standard) and others who were knowledgeable on the standard to attempt to resolve the problem or to discover how others have interpreted it. The resulting emulation, therefore, may not be consistent with a

future release of the standard when these points are clarified.

The following are the known clarifications and differences between this emulation and MIL-STD-1750, dated 21 February 1979:

- None of the input or output instructions marked as OPTIONAL in the standard were implemented.
- The only bits in the fault register that are used are bit 5 (input command used with an output code or an output command used with an input code) and bit 9 (illegal opcode). All other bits either depend on optional features not implemented (e.g., parity, protected memory, checking for illegal addresses) or are undefined.
- Handling of the instruction counter is inconsistent in the standard. In the definitions section of the standard, the IC is defined as holding "the address of the next instruction to be executed" (Ref 3: 1). This implies that the IC should be incremented immediately following the fetch cycle and before the fetched instruction is actually executed. In the MOV instruction (move multiple words memory to memory), the standard specifies that the IC points to the current instruction (the MOV instruction) until the last word is transferred. In the definition of the instruction counter relative addressing mode (ICR) used for branch statements, the standard says "the content (sic) of the

instruction counter (i.e., the address of the current instruction) is added to the sign extended 8-bit displacement field of the instruction" (Ref 3: 16). This implies that the IC is not incremented until after the address computation is made for the branch statement, assuming that by "address of the current instruction" the standard is referring to the address of the branch statement currently being executed. In this emulation, the IC is always incremented immediately after the instruction fetch and before the instruction is decoded and executed. The MOV instruction is treated as a special case to conform to the standard. the displacement in an instruction counter relative instruction is added to the address of the instruction following that instruction, based on a still-uncertain clarification.

- On start-up of the emulation, all interrupts are masked (i.e., inhibited) except for interrupts 0 and 1, which cannot be disabled. This allows the programmer to set up the interrupt handlers without being interrupted.
- The enable interrupts output command enables the interrupts "after execution of the next instruction" (Ref 3: 129). This allows one instruction to be executed following the enable interrupt instruction and before interrupts are actually enabled, allowing, for example, the microprogram to enable interrupts

as the last instruction in an interrupt service routine and complete the return from interrupt instruction (LDST) before other interrupts can be acknowledged.

- The disable/enable interrupt commands to the Am2914 interrupt control unit are not used to implement the MIL-STD-1750 disable/enable interrupts, since the two highest priority interrupts cannot be disabled. Instead, the mask register is used to accomplish the same effect.

Emulation Hardware

The hardware additions to the basic MIME are contained on an auxiliary circuit board (to be referred to as the aux board) installed at the AUX module position. This board has available to it all the user definable commands, functions, and test condition signals derived from those fields in the microword, in addition to the data bus and various other control signals. This makes it possible to remove one aux board (for the MIL-STD-1750 emulation, for example) and replace it with a different aux board that fulfills the hardware requirements for another program or emulation.

The hardware functions realized on the MIL-STD-1750 aux boards are (see Figure 5):

- Additional general purpose registers
- Timers
- Status word
- Fault register
- Condition flags

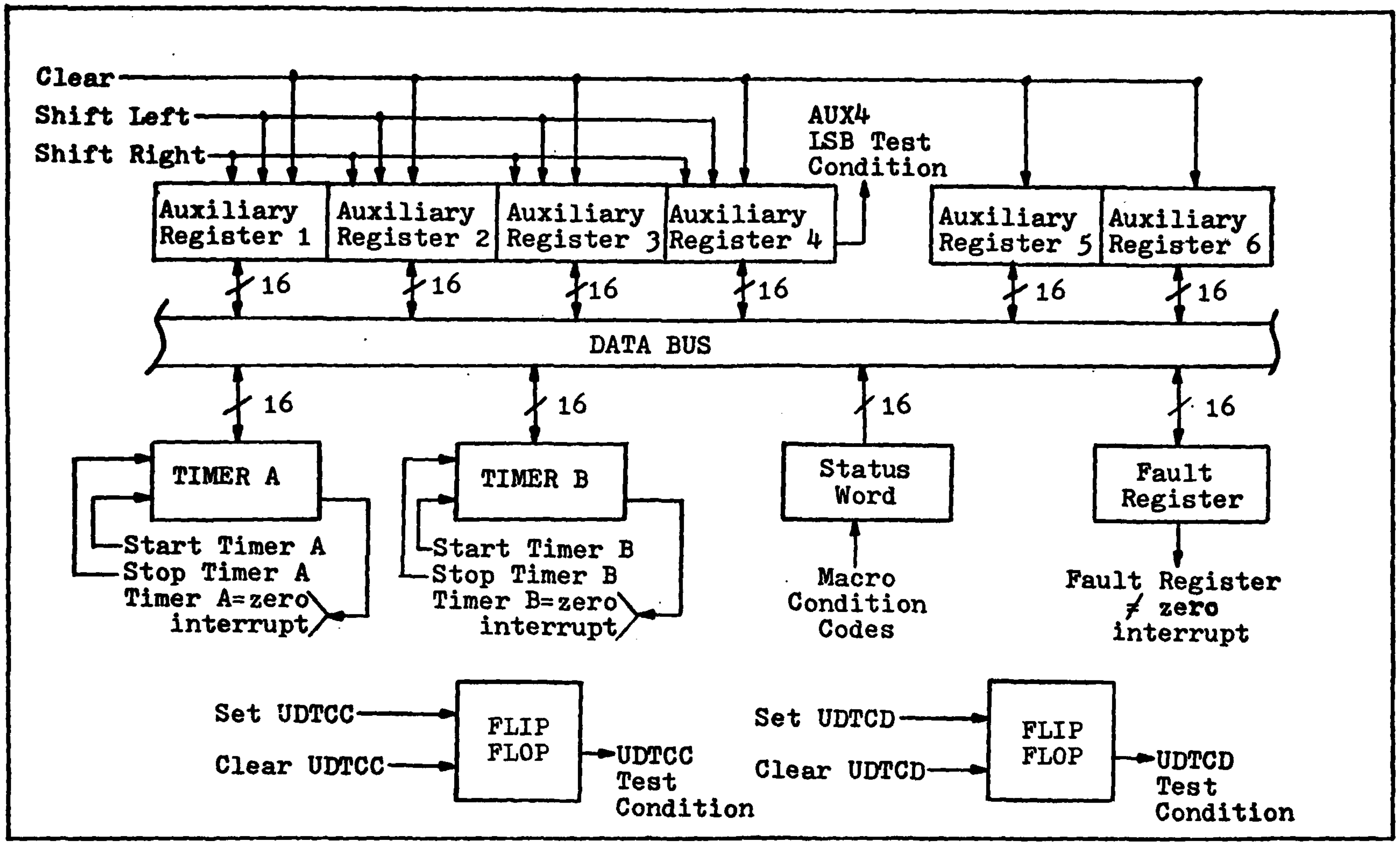


Figure 5. MIL-STD-1750 Auxiliary Board Block Diagram

Registers. Six general purpose registers, in addition to the sixteen associated with the MIME ALU, were implemented on the aux board. These would find their primary utility in doing double precision, floating point, and extended floating point operations.

Timers. Two timers, as defined in MIL-STD-1750, were implemented. Timer A increments every 10 microseconds while timer B increments every 100 microseconds.

Status Word. The status word in MIL-STD-1750 is a recombination of the carry, negative, and zero macro condition codes in MIME into the carry, positive, zero and negative bits of the status word. These only occupy the most significant four bits of the sixteen bit register. The remaining twelve bits are available for use should they be defined in later releases of MIL-STD-1750.

Fault Register. The fault register is used to signal machine error conditions and generates a machine error interrupt if any of its bits are set.

Condition Flags. Three additional condition flags are available. Two can be set and cleared on command from the microprogram through the AUX COMMAND field. The third one is connected to the least significant bit of auxiliary register four. All can be used as a test condition for a conditional microstatement.

Aux Board Interface to the Microword. The aux board is controlled by the AUX COMMAND and AUX FUNCTION fields in the microword (see Appendix A). The AUX COMMAND field (bits 23 - 20) has various commands for the aux board and is enabled by an "F" in the COMMAND field of the microword (bits 15 - 12). Note that the auxiliary commands share their field in the microword with the interrupt control commands. The interpretation of this field depends on which is enabled from the command field.

The AUX FUNCTION field (bits 19 - 16) is used to select which register on the aux board is accessed on a read from or write to the aux board. A read of the aux board is signified by a "5" in the BUS SOURCE field (bits 11 - 8) and a write to the aux board by a "D" in the BUS DESTINATION field (bits 7 - 4).

The auxiliary conditions are selected using the CCU TC MUX field (bits 59 - 56).

Emulation Software

The emulation microcode consists of five main parts:

- Instruction fetch
- Instruction decoding
- Address mode interpretation
- Instruction execution
- Interrupt handling

Figure 6 gives a general flow chart of the organization of the emulation.

The instruction fetch includes reading a new instruction from the memory location pointed to by the program counter and incrementing the program counter to point to the next memory location. Instruction decoding determines the particular

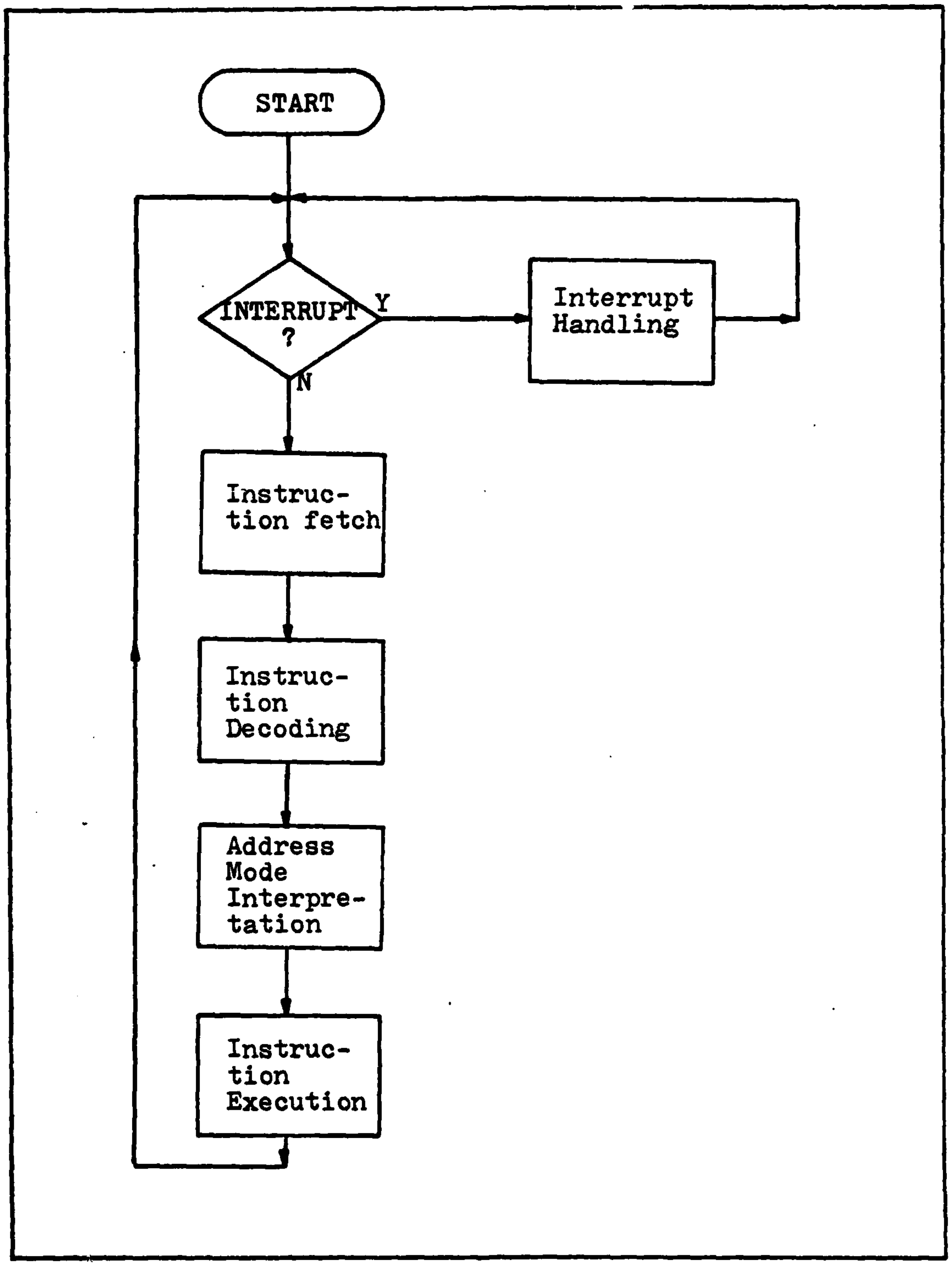


Figure 6.

MIL-STD-1750 Emulation Block Diagram

type of instruction involved, based on its opcode, and transfers to the appropriate section of microcode that processes the instruction. The first step involved in instruction processing is to determine the addressing mode being used and to identify the operands to be used for the instruction execution. The execution phase then uses these operands and performs the operations specified by the instruction type.

Interrupts may occur at any time but are only recognized by the microprogram prior to the instruction fetch phase. If an interrupt is pending, it is processed before the instruction fetch takes place. Processing the interrupt involves saving the current machine status and transferring to a user-defined interrupt routine. A more detailed discussion of the component parts of the emulation software can be found in Appendix B.

Emulation Results

The emulation microcode implements the complete MIL-STD-1750 instruction set, except the extended precision floating point arithmetic instructions. These were not implemented because of the time constraints of this investigation. However, the instructions that were translated into microcode comprise a representative sample of the types of instructions available on many current minicomputers. These include single/double precision integer arithmetic and logical operations, floating point operations, bit/byte manipulations, subroutine mechanisms, input/output operations, and control instructions.

The number of microinstructions (clock cycles) needed to perform each of the MIL-STD-1750 instructions is tabulated in Appendix B.

Table III contains a more detailed breakdown of the microinstruction counts for three representative MIL-STD-1750 instructions. The microroutines for these same instructions are presented in detail in Appendix B. The microinstruction counts are converted to execution times and are compared with the execution times for similar instructions belonging to other computers.

Although the MIME execution times are significantly slower than those of the others, certain considerations should be understood:

- The intent was to develop clear, straightforward microcode implementations of the MIL-STD-1750 instructions. It was not intended to maximize the speed of execution or to minimize the amount of control store used.
- Multiplication (and division) and floating point normalization are two of the largest consumers of execution time in this emulation. As can be seen by reference to Table III, these operations comprise the bulk of the total execution time for the floating point multiply instruction. These operations could be implemented in hardware using available integrated circuits to derive a significant improvement in instruction execution times. This could be accomplished either by providing a hardware multiplier on the aux board or by redesigning

Table III

Example Instruction Execution Breakdown

Instruction Execution Phase	Microinstruction Counts[1]		
	Single Precision Integer Add	Single Precision Integer Compare	Floating Point Multiply
Instruction Fetch	5	5	5
Address Decoding	13	13	13
Operand Fetch and Set-up	6	5	26
Perform Operation	2	9	660
Handle Exponent			5
Handle Mantissa			313
Normalize Result			333
Miscellaneous			9
Total[2]	25	31	699
Execution Time (with 2 microsecond clock) in microseconds	50	62	1398
LSI-11 with KEV11 Extended Arithmetic Unit (Ref. 12:B-1 to B-4) in microsec.	17.15	16.5	178.5[3]
AN/AYK-15A Proposal by Westinghouse (Ref. 7: Vol. 4) in microseconds	1	1.25	6

Note 1 - Maximum number of MIME clock cycles.
Note 2 - Total may not be the sum of the listed component parts, since each value is a worst-case count.
Note 3 - Mantissa multiplication takes 121.1 microseconds. Normalization takes 18.9 microseconds.

the ALU board around the more advanced Am2903 "Super Slice" processor.

- MIME currently is using 2708 EPROM's that have a 450 nanosecond access time as control store. Although the MIME system clock is running at a frequency of 1 MHz, the wait circuitry required to allow for reliable control store access cancels every other clock pulse. Thus, the effective system clock period is 2 microseconds per microinstruction. MIME is capable of operating at a faster rate if the control store and wait circuitry were made compatible with the faster speed.

Summary

This chapter has presented the MIL-STD-1750 emulation design. Discussions of the emulation hardware and emulation software were given along with a brief discussion of the emulation results.

VI. Results, Recommendations, and Conclusion

The objectives of this investigation were to add the necessary enhancements to MIME to allow its use by a novice programmer, and to provide a machine capable of executing the MIL-STD-1750 instruction set. Meeting these objectives required the attainment of two goals:

- Enhancement of MIME hardware, software, and documentation to provide usable tools for the programmer.
- Microprogramming of the MIL-STD-1750 instruction set.

This chapter discusses the investigation results, provides recommendations for further research, and presents the conclusions reached.

Results

The following paragraphs discuss the primary results of this effort.

MIME Hardware Enhancement. Initially, the MIME hardware was essentially complete, but not debugged. No machine instructions had actually been executed by the machine. During this phase of the project, the hardware was first made to run reliably. This was done with much assistance from one of the original MIME designers.

In parallel with that activity, the MIME architecture was compared to the architecture required by the MIL-STD-1750 minicomputer instruction set. This was done to determine what hardware additions would be required to emulate that instruction set on MIME. Several hardware features were found to be lacking in the MIME design. These features were integrated into MIME through a newly designed, user-definable interface. Chapters II and III discuss this facet of the project, while Appendix A provides a User's Manual for the resulting MIME hardware.

Microprogramming Language and Translator. Previous to this investigation, all microprogramming of the MIME had to be done by hand-assembling a sequence of 64-bit microwords, then loading this microcode into control store through a keyboard on the front panel. This was a totally untenable task for the development of non-trivial microprograms, since errors were difficult to find and correct.

To remedy this situation, a MIME microprogramming language was defined. This language allowed the microprcgrammer to express microroutines in free-flowing statements and mnemonics. A translator was created to convert this text into MIME microcode while doing error and consistency checks on the program. The translator can provide both MIME microcode and the code for mapping a target instruction set's opcodes into the corresponding microcode routines. The code thus generated may be programmed into EPROM's and inserted into the control store and mapping PROM sockets in MIME. The language and translator are discussed in Chapter IV, a reference manual is included as Appendix C, and the information required to program EPROM's from the translator output is presented in Appendix N.

Monitor. The translator was used in the development of a monitor program called the MIME Modest Monitor (MIME/MM). MIME/MM allows the user to load and dump memory and registers through a terminal. This capability speeds the evaluation and debugging of both macro and microprograms. The monitor is discussed in Chapter IV and is fully described in Appendix D.

MIL-STD-1750 Emulation. A major subset of the MIL-STD-1750 instruction set was microprogrammed. The extended precision floating point arithmetic operations were not implemented due to time constraints, but all of the other instructions were microcoded. This included all floating point and single/double precision integer arithmetic and logical instructions as well as the bit manipulation, register and memory manipulation, input/output, and control types of instructions.

The MIL-STD-1750 instruction set was used to help define the requirements for the user-defined additions to MIME, as well as to test hardware and software enhancements. Furthermore, the emulation helps fill the requirement of the Aeronautical Systems Division Computer Engineering Branch (ASD/ENAIB) to have a "test-bed" for evaluation of the standard instruction set. A discussion of the emulation structure and rationale is advanced in Chapter V, and Appendix B provides a reference manual.

Documentation. An underlying task during all portions of this investigation was to update existing documentation and provide new documentation as needed. In addition to the appendices already mentioned, there are wire-run lists, schematic diagrams and parts lists for all hardware and source code listing for all software in Appendices E through M.

Recommendations

This section advances several recommendations for further investigations involving MIME and/or the MIL-STD-1750 emulation.

Computer Based Development System. For MIME to be truly useful as a research/teaching tool requires its integration into a development system. Such a system might include:

- A computer based controller with user I/O and mass storage capabilities.
- Transportation of the translator program to the controller. This would allow rapid changes to be made in microcode with the results stored in mass storage. The controller should be capable of running an implementation of Pascal.
- A control store manager to act as the interface between control store and the MIME I/O. This would allow microcode to be transferred from control store to the controller (and thence to mass storage), and vice-versa, through the MIME I/O ports.
- Writable control store to allow microcode to be loaded by the control store manager.
- An executive program to be run on the controller to interact with and expand the capabilities of the MIME/MM and the control store manager (ideally in a higher order language for ease of implementation/use).

- Cross-translator(s) for target machine assembly or higher order languages to run on the controller.

A possible configuration for a development system is shown in Figure 7. Writable control store could be easily implemented by designing a new CS2 board with RAM instead of EPROM. The control store manager might be either a hardware design, or it could be realized as a microprocessor-based subsystem.

ALU Redesign. Since the MIME was designed, Advanced Micro Devices, Incorporated has introduced the Am2903 "Super Slice" and the Am2904 Status and Shift Control Unit, which contains most of the logic required to surround the ALU chips (Ref 8: 2 - 30 to 2 -55). The Am2903 has built-in multiply and divide logic, has facilities for an expanded register file, and has features to more efficiently handle floating point operations, particularly normalization. Since, as discussed in Chapter V, the multiply and normalize segments comprise the majority of the execution time of a floating point multiply operation, reducing these segments' execution times would greatly improve the overall performance of the operation. The MIME ALU might be significantly improved by redesigning it around these new circuits. It is recommended that the feasibility of such a redesign be investigated.

MIL-STD-1750 Extensions. The extended precision floating point instructions of MIL-STD-1750 should be implemented to complete this emulation. This would mostly require driver routines to call upon the existing floating point routines and

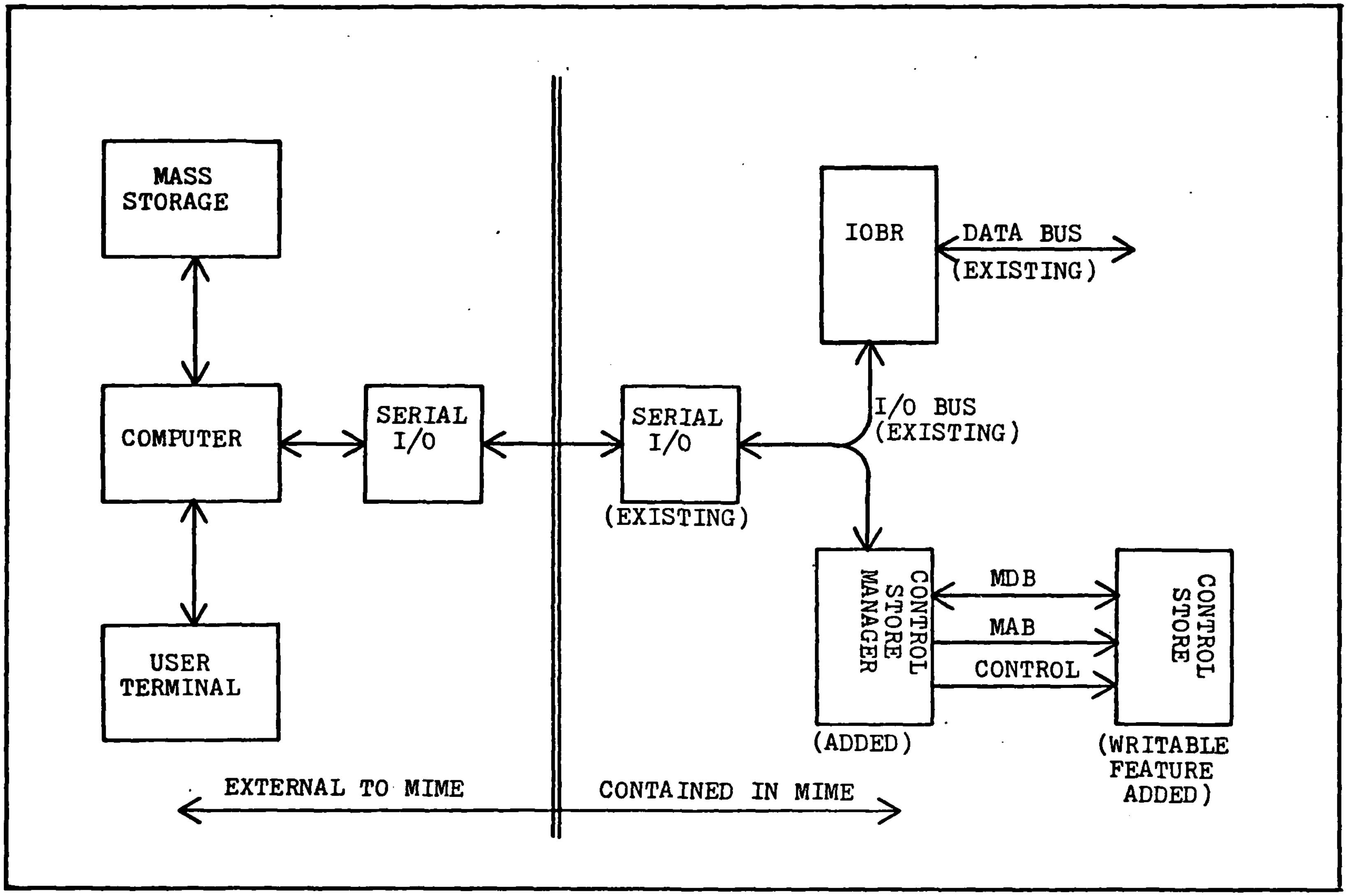


Figure 7. Computer-based Development System

extend their use. Also, it is recommended that the feasibility of implementing such optional features as memory protection, parity checking and activation of additional Fault Register bits be evaluated.

Microcoded Self-diagnostics. As an aid to assuring the proper operation of MIME, a series of self-diagnostic routines could be generated. These routines could be activated upon start-up and could check the status of memory, registers, buses, etc., reporting any faults found to the operator. Such routines would be best implemented in microcode since only microcode has access to all of MIME's processor elements.

Conclusion

The objectives stated in Chapter I have been essentially met. The implementation of the MIL-STD-1750 instruction set has demonstrated that MIME is in fact capable of executing a minicomputer instruction set. Furthermore, since the MIL-STD-1750 implementation is microprogrammed, future modifications and enhancements to the instruction set may be done with relative ease. Since the instruction and register requirements of many processors are subsets of the MIL-STD-1750 requirements, the authors feel assured that any of those processors could be emulated by MIME. Any additional hardware that might be needed to accomplish such emulations could be incorporated by making use of the user-definable functions and commands added during this investigation.

The software development tools which were developed during this investigation were demonstrated to make the microprogramming task an easier one. The user is relieved not only of the tedium of hand assembling large amounts of microcode, but of the effort and frustration of discovering and correcting minor errors in microword formatting and program consistency.

Finally, the documentation amassed during this effort provides the user with a complete, consistent set of references. Information is included to allow future users to utilize MIME's hardware features, to correctly microprogram MIME with minimum wasted effort, and to make extensions or modifications to the present machine. Thus, with these three elements - usable hardware, usable software tools, and usable documentation - MIME has become a viable research/pedagogical system.

Bibliography

1. Purvis, Richard E. and Ronald D. Yoho. MIME: Microprogrammable Minicomputer Emulator. Unpublished Thesis. Wright-Patterson Air Force Base: Air Force Institute of Technology, 1978.

2. Abd-alla, Abd-elfattah M. amd Arnold C. Meltzer. Principles of Digital Computer Design, Vol 1. Englewood Cliffs: Prentice Hall, 1976.

3. MIL-STD-1750 (USAF). Airborne Computer Instruction Set Architecture. Washington: Department of Defense, 21 February 1979.

4. Iftekhar, Saleem and John Pennett. Design of an RS232 Interface for the Microprogrammable Minicomputer Emulator (MIME). Unpublished laboratory report. Wright-Patterson Air Force Base: Air Force Institute of Technology, 1978.

5. Thourot, Frederick G. Design of the Processor for Software Compatible Avionics Computer Family. Unpublished Thesis. Wright-Patterson Air Force Base: Air Force Institute of Technology, 1977.

6. Airborne Digital Avionics Module (ADAM). U. S. Air Force contract F33657-74-C-0727. Contracted with McDonnell Douglas Astronautics Company, St. Louis, MO.

7. Proposal for AN/AYK-15A Processor Development. Response to RFP F33615-79-R-1733. Westinghouse Electric Corp., Baltimore, MD, 1979.

8. The Am2900 Family Data Book with Related Support Circuits. Sunnyvale, CA: Advanced Micro Devices, Inc., 1978.

9. Madnick, Stuart E. and John J. Donovan. Operating Systems. New York: McGraw-Hill, 1974.

10. Hwang, Kai. Computer Arithmetic Principles, Architecture, and Design. New York: John Wiley and Sons, 1979.

11. Intel 1977 Data Catalog. Santa Clara, CA: Intel Corporation, 1977.

12. Microcomputer Processors. Maynard, MA: Digital Equipment Corporation, 1978.

Vita

Thomas R. Hoyt was born on 5 December 1947 in Kalamazoo, Michigan. A 1966 graduate of Gull Lake Community Schools, Hickory Corners, Michigan, he attended Northwestern University, Evanston, Illinois, from which he received the degree of Bachelor of Science (Mechanical Engineering) in June 1971. He received a commission in the USAF through Officer Training School in October 1971, and was assigned to Williams AFB, Arizona for Undergraduate Pilot Training. He received the aeronautical rating of Pilot in October 1972, and served as an RF4C aircraft commander at Zweibrucken AB, Germany until December 1976. He then was assigned to the Recon/Strike System Program Office at Wright-Patterson AFB, until being assigned to the School of Engineering, Air Force Institute of Technology, in June 1978.

Permanent Address: 9827 East DE Avenue
Richland, MI 49083

Vita

Dean A. Myers was born on 3 July 1950 in Los Angeles, California. A 1967 graduate of Westchester High School, Los Angeles, California, he attended Loyola University of Los Angeles for three years, after which he entered the United States Air Force Academy. After graduating from the Academy in June, 1974 with a Bachelor of Science in Electrical Engineering, he served as a Computer Systems Analyst at Headquarters, Air Force Logistics Command, Wright Patterson AFB, Ohio until entering the School of Engineering, Air Force Institute of Technology, in June, 1978.

Permanent Address: 6406 West 84th Street
Los Angeles, CA 90045

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/79-11	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MIME: MICROPROGRAMMABLE MINICOMPUTER EMULATOR, PHASE II		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Thomas R. Hoyt and Dean A. Myers, Capt, USAF Capt, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Computer Engineering Branch (ASD/ENAIB) Aeronautical Systems Division Wright Patterson AFB, OHIO 45433		12. REPORT DATE December, 1979
		13. NUMBER OF PAGES 92
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

[illegible]

J. P. Hipps, Major, USAF
Director of Public Affairs

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microprogramming
Emulation
Bit Slice Architecture
AMD 2900
MIL-STD-1750

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes software and hardware enhancements made to an existing educationally oriented microprogrammable minicomputer (MIME) that was implemented using the Am2900 series of integrated circuits. Hardware changes included extensions of the machine architecture by incorporation of user-definable fields in the microword, corresponding to user-definable capabilities in the hardware. A microprogramming language was developed for the machine, as well as a translator. The instruction set of MIL-STD-1750, Airborne Computer Instruction Set Architecture was microprogrammed to demonstrate the utility of the translator and the computer. MIL-STD-1750 peculiar hardware was interfaced through the user-definable portion of the architecture.

The results showed that a minicomputer instruction set could be successfully emulated using MIME. The final system of hardware, software, and documentation provides a valuable educational tool for studies of microprogramming, emulation, and computer control.